INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.(1) 39 63 55 11

Rapports de Recherche

N° 1402

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

TOWARDS A META SIMPLIFIER

Christèle FAURE

Avril 1991

RR-1402

Programme 2
Christèle Faure[1]
faure@psyche.inria.fr

Vers un méta simplificateur

Dans les systèmes de calcul formel actuels, le mécanisme de simplification est faiblement paramétré. Si la définition d'expression plus simple ne satisfait pas l'utilisateur, il n'a que peu de moyens de la modifier, et même pas du tout en ce qui concerne les opérateurs courants ($+, *, -, abs, log...$). Il est alors obligé de redéfinir intégralement de nouveaux opérateurs (avec de nouveaux noms) correspondant à ses besoins. De notre point de vue, un simplificateur doit être complètement personnalisable et ne peut l'être que si les connaissances algébriques utilisées sont séparées du processus de simplification.

Dans ce rapport, nous présentons un modèle de simplificateur composé de deux parties : la **base** qui contient toute la connaissance, et le **moteur** qui effectivement simplifie les expressions. Tout d'abord, nous décrivons deux catégories de connaissances de base : les équations et les fonctions de calcul, ainsi que les deux processus qui les manipulent. Comme un simplificateur est toujours connecté à un système, nous définissons ensuite les interactions entre un simplificateur et son système hôte.

Towards a meta simplifier

In distributed computer algebra systems, the simplification process modifies each input expression to fit a certain definition of "simplicity". This definition is given (through the written simplifier) by the system designer and not by the user. However this definition may not correspond to the user's notion of simplicity. We consider that simplifiers cannot be customizable as far as the simplification knowledge is encoded in the simplification functions.

In this paper, we present a model of simplifier divided in two parts : the **base** that contains all the knowledge and the **engine** which really computes simplification. First, we describe two basic kinds of knowledge (equations, evaluation functions), and their two associated processes. Since a simplifier is always connected to a system, we try then to define the interactions between a simplifier and a host system.

[1]INRIA, Centre de Sophia-Antipolis, 2004 route des Lucioles, 06565 Valbonne Cedex, ou Université de Nice-Sophia Antipolis, Dépt. de Mathématique, Parc Valrose, 06034 Nice Cedex, France.

1 Introduction

In a computer algebra system, the simplification process modifies each input expression to fit a certain definition of "simplicity". This definition depends on the habits of the user (for instance the notations he is used to), and the further use of the result. When a system designer implements a simplifier, he tries to make this definition more precise (in particular for the usual operators like $+, -, *, \cdot$), and he assumes that it is the standard one. However this definition may not fit the user's one, therefore the use of the computer algebra system is less attractive or completely irrelevant for a given purpose. Therefore the design of an easy customizable simplifier is an important issue for making computer algebra systems more usable.

To make simplifiers completely customizable, we think that the "knowledge" involvedmust be separated from the simplification process itself. For example, the simplification of the expression $((2 + -2) + 3) + 1$ must give 4. The knowledge involved in this simplification consists of two rewrite rules : $X + zero \rightarrow X$ and $-X + X \rightarrow zero$, an equation : $X + Y = Y + X$ (commutativity) and the evaluation function that adds two integers. First the system simplifies the sub-expression $2 + -2$ to *zero* by using the second rule modulo the equation, the new expression $zero + 3$ is simplified to 3. The equation and the two rules have been used by the system to evaluate $3 + 1$ in 4. More generally, the same kind of knowledge is needed : equations, rewrite rules and evaluation functions. In this description, we don't take into account functionalities such as substitution, search of specific sub-expressions.

This new approach enable the user to give all the equations, the rewrite rules and the evaluation functions he needs, in order to get output expressions the way he wants them. This information is structured around the notion of operator, and stored in a **base**. "All information about each operator is grouped, but the information about each domain[2] is scattered", this seems much more flexible than a domain oriented base, even if it is a less compact description.

The base described before is the first part of the meta-simplifier, the second part, called the **engine** effectively simplifies expressions. This process splits in two parts : the first one manages the equations and rules and the

[2]A domain is a representation of a type and the operators related to it, like in Scratchpad[JSW86].

second one manages evaluation functions.
Clearly, when nothing is stored in the base the simplifier does nothing, the output expression is the same as the input expression.
To define an operator the user gives its name, the equations and the evaluation functions related to it.

The simplifier described in this paper has been implemented in connection with the computer algebra system Sisyphe[3]. We give a general approach and we briefly analyze the implications of the interaction between a simplifier and an host system.
In the following sections, we describe how a new operator can be defined in the base, and how the engine uses those definitions to simplify expressions.

2 Defining an operator by equations

Each existing system gives a way of declaring equations, either by attaching equationnal properties to operators (for instance, in Macsyma[The83], one declares properties by $declare(f, linear)$,$declare(g, multiplicative)$), or by declaring rewrite rules (in Macsyma, an example of declaration is $tellsimp(D[x](A, B), B(x) * diff(A(x), X) - A(x) * diff(B(x), x))$, where x is a pattern variable and X is the differentation variable). A rewrite rule is nothing but an oriented equation.

Rewrite rules are operationally more intersting than equations. But among usual equations, $X + Y = Y + X$ (commutativity) cannot be oriented to make a rule, so we deal with it through a property. All orientable equations can be declared in the form of rules, but we have noticed that the same sets of rules are given several times with different operator names. For example the set of rules $\{X + -X \rightarrow zero, X + zero \rightarrow X, -(-X) \rightarrow X\}$ and $\{X * inv(X) \rightarrow one, X * one \rightarrow X, inv(inv(X)) \rightarrow X\}$ are similar if we introduce three parameters, which are $(+, -, zero)$ in the first case and $(*, inv, one)$ in the second one.
In order to factorize those similar declarations, we defined properties equivalent to them. Another advantage of grouping rewrite rules in properties is given by the possibility of writing optimized algorithms, which do the same work as rewrite rules. All the equations involved in the definition of useful mathematical structures (group, ring, field) are then defined by properties.

[3]Developed by INRIA and University of Nice.

In the following, we will describe the action of the engine at the first level using the properties of the root operator of the term.

2.1 Simplification modulo one basic property

This section describes the simplification of a term modulo one property of its root operator. We choose to define a set of general properties in order to have many possible applications. These properties allow the user to define usual mathematical structures such as group, ring, field ... then they are called **basic properties.**
So an operator could be : associative, g-symmetric, idempotent, nilpotent, n-homogeneous,
or could have : a unity element (which is a constant function), a symmetric function, an absorbing element (constant function).

Each property is taken care by a specific algorithm which takes a term as input and simplifies it modulo the property. To simplify a term modulo a property, the system "asks" the base whether the operator has this property, if the answer is yes, it applies the corresponding algorithm.

We decide to simplify associative term in their flattened form : $(a + (b + c)) \rightarrow (a + b + c)$. In order to simplify a g-symmetric term t_1 with respect to some operator g, we sort[4] the first-level sub-terms of t_1 and get t_2, if the signature of the permutation is -1 then the result is $g(t_2)$ otherwise it is t_2. For example, commutativity is a consequence of the previous property with $g = identity\text{-}function$.

The previous choices for simplified forms modulo associativity and g-symmetricity imply the following definition for a **generalized sub-term** of the term $op(a_1, ..., a_n)$ (after simplification modulo associativity and g-symmetricity). Its operator is op and its arguments are :

- in associative/g-symmetric context : an ordered sub-set of $\{a_1, ..., a_n\}$ (for example $a + c$ is a generalized sub-term of $a + b + c$),
- in associative context : a sub-sequence of $a_1, ..., a_n$ (for example $m_1 \cdot m_2$ is a generalized sub-term of $m_1 \cdot m_2 \cdot m_3$),
- otherwise : $a_1, ..., a_n$.

[4]This algorithm is parametrized by orders over integers, strings, polynomials ...

One can notice that a generalized sub-term is simplified modulo associativity and g-symmetricity.

All our other properties are equivalent to rewrite rules, therefore the action of the simplifier can be described by the applied rules :

Assertion	rule
$projector(f)$	$f(f(x)) \rightarrow f(x)$
$involutive(f)$	$f(f(x)) \rightarrow x$
$idempotent(f)$	$f(x,x) \rightarrow x$
$nilpotent(f,n,h)$	$\underbrace{f(f(f...f}_{n}(x))) \rightarrow h$
$has\text{-}zero\text{-}l(f,zero)$	$f(zero,x) \rightarrow x$
$has\text{-}zero\text{-}r(f,zero)$	$f(x,zero) \rightarrow x$
$has\text{-}abs\text{-}l(f,ab)$	$f(ab,x) \rightarrow ab$
$has\text{-}abs\text{-}r(f,ab)$	$f(x,ab) \rightarrow ab$
$has\text{-}symmetric\text{-}l(f,h)$	$f(h(x),x) \rightarrow zero$
$has\text{-}symmetric\text{-}r(f,h)$	$f(x,h(x)) \rightarrow zero$
$ext\text{-}op(f_1,f_2)$	$f_1(s_1,f_1(s_2,t_2)) \rightarrow f_1(f_2(s_1,s_2),t_2)$
$n \times m\text{-}morphism(f,g,h)$	$f(x_1,...,g(x_i^1,...x_i^m),...,x_n) \rightarrow$ $h(f(x_1,...,x_i^1,...,x_n),...f(x_1,...,x_i^m,...,x_n))$
$n\text{-}homogeneous(f,g,h)$	$f(x_1,...,g(a,x_i),...,x_n) \rightarrow$ $h(a,f(x_1,...,x_i,...,x_n)$

Example : The operator + can be defined by the following assertions : $associative(+)$, $g\text{-}symmetric(+,Fid)$[5], $has\text{-}zero(+,zero)$, $has\text{-}symmetric(+,-)$. The term $n+y+(-n+x)$ is simplified to $n+y+-n+x$ by associativity, $n+x+y+-n$ by commutativity, $x+y+zero$ because of the symmetric, $x+y$ because of the zero.

2.2 Simplification modulo a set of basic properties

One can attach a set of properties to an operator, we describe the simplification of a term modulo such a set. Simplifying a term modulo a list of properties is equivalent to rewriting a term modulo a set of rules, it implies the choice of a strategy. But our case is much easier than the general rewriting problem because we know all the authorized properties in advance and

[5]Fid is the name of the identity function, so this declaration is equivalent to commutativity.

we have results about the canonicity([Hul80]) of sets of rules equivalent to those properties.

We define an order over those properties, it's a partial order because we only sort coherent lists of properties. Given a term and a sequence $p_1, ..., p_n$ of properties (sorted), we use those properties from 1 to n. Each p_i is used every time it's possible, the output term is still simplified modulo $p_1, ..., p_{i-1}$[6], then the system has only to simplify it modulo $p_{i+1}, ..., p_n$.
The most part of our authorized properties are mutually exclusive and the only ordered one are : $ext\text{-}op < idempotent < has\text{-}symmetric < has\text{-}zero < has\text{-}abs$.

Simplifying a term modulo $p_1 < ... < p_n$ and associativity(A) or g-symmetricity(C), consists in simplifying it first modulo associativity, second modulo g-symmetricity, then modulo (A, C, p_1) ... (A, C, p_n). The two properties A, C cannot be "forgotten" in the following simplifications because they induce the definition of generalized sub-terms. For example if $-$ is the symmetric of $+$ with $zero$ for zero, we want $n + x + y + -n$[7] to become $x + y$. This is only possible if the system knows that $+$ is AC, then $n + -n$ is a generalized sub-term for $n + x + y + -n$ and the system computes $x + y + zero$, $y + zero$ is a generalized sub-term for $x + y + zero$ and the system computes $x + y$.

In the case where the list of properties associated to the root of the input term contains a sequence s of ordered properties and a list l of other uncomparable properties, the system simplifies the term modulo l and then modulo s. Indeed, our uncomparable properties can change the root operator of the term, so there is no need for simplifying the input term if it is completely changed (even the root operator is changed). To simplify a term modulo the previous list $l = (l_1, ..., l_n)$, the system acts as if it was a list of rules, it uses each l_i every time it's possible with i from 1 to n and then do it again while the term changes.

For example, $*$ can be defined by :
$associative(*)$, $g\text{-}symmetric(*, Fid)$, $has\text{-}zero(*, one)$,
$has\text{-}symmetric(*, inv)$, $has\text{-}abs(*, zero)$, $2 \times 2\text{-}morphism(*, +, +)$[8],

[6]This property is given by the order. For example, using "has a zero" before "has a symmetric" isn't valid because the use of "has a symmetric" can generate a "zero", then we have to use "has a zero" a second time.

[7]The order between involved terms is $n < x < y < -n < zero$.

2-$morphism(*, -, -)$.
The term $a * (c + inv(a) + one)$ is simplified :
modulo associativity to $a * (c + inv(a) + one)$,
commutativity to $a * (c + inv(a) + one)$,
$2 \times$ 2-morphism(*,+,+) to $a * c + a * inv(a)$,
then the term $a * inv(a)$ is simplified
modulo symmetric to one,
and the complete term $a * c + a * inv(a)$ is simplified to $a * c + one$.

2.3 Extended example

This example deals with euclidian geometry in R^3, the involved operators are : $+, -, zero, *, one, inv, \cdot, mul, t, tilde$. Where $\cdot$ is the product between matrices, mul the product between a scalar and a matrix, t is the transposition operator and $tilde$ is the operator such as $\tilde{X} \cdot Y = X \wedge Y$. Constants are introduced : Id (the identity matrix), $Mzero$ (the matrix zero), Fid (the identity function), $X0, Y0, Z0$ (the unit vectors over the axes).

The following package of properties and rules is a sample which can be useful in mechanics. More precisely, those properties associated to the set of rules is the one chosen in the simplifier of Gemmes[CG89] which is a program written is Maple and developped by Aerospatial. Properties of the operators $-, inv, \cdot, mul, t, tilde$:

Name	Assertions	Name	Assertions
t	$involutive(t)$	mul	$has\text{-}zero\text{-}l(mul, un)$
	1-$homogen.(t, mul, mul)$		$has\text{-}zero\text{-}r(mul, Id)$
	$1 \times$ 2-$morphism(t, +, +)$		$has\text{-}abs\text{-}l(mul, zero)$
	$1 \times$ 1-$morphism(t, -, -)$		$has\text{-}abs\text{-}r(mul, zero)$
inv	$involutive(inv)$		$ext\text{-}op(mul, *)$
$-$	$involutive(-)$	$\cdot$	$associative(\cdot)$
	$1 \times$ 2-$morphism(-, +, +)$		$has\text{-}zero(\cdot, Id)$
$tilde$[9]	$1 \times$ 2-$morphism(tilde, +, +)$		$ext\text{-}op(mul, *)$
	1-$homogen.(tilde, mul, mul)$		2-$homogen.(\cdot, mul, mul)$
			$2 \times$ 2-$morphism(\cdot, +, +)$

For the operators + and * we keep the declarations described in the previous sections (respectively 2.1 and 2.2).

[9] We denote $tilde(X)$ whether $\tilde{X}$ or $tilde(X)$.

Rules :

$\widetilde{Mzero}$	$\rightarrow$	$Mzero$	$t(Id)$	$\rightarrow$	Id
$\widetilde{m} \cdot m$	$\rightarrow$	$Mzero$	$rot(ax,\theta) \cdot ax$	$\rightarrow$	ax
$-mul(s,m)$	$\rightarrow$	$mul(-s,m)$	$rot(ax,\theta) \cdot t(rot(ax,\theta))$	$\rightarrow$	Id
$mul(s,-m)$	$\rightarrow$	$mul(-s,m)$	$\widetilde{Y0} \cdot X0$	$\rightarrow$	$-Z0$
$\widetilde{X0} \cdot Z0$	$\rightarrow$	$-Y0$	$\widetilde{Z0} \cdot X0$	$\rightarrow$	$Y0$
$\widetilde{X0} \cdot Y0$	$\rightarrow$	$Z0$	$\widetilde{Z0} \cdot Y0$	$\rightarrow$	$-X0$
$\widetilde{Y0} \cdot Z0$	$\rightarrow$	$X0$	$t(rot(ax,\theta)) \cdot rot(ax,\theta)$	$\rightarrow$	Id

$$rot(Z0,\theta) \cdot X0 \rightarrow mul(sin(\theta),Y0) + mul(cos(\theta),X0)$$
$$rot(Z0,\theta) \cdot Y0 \rightarrow mul(cos(\theta),Y0) + mul(sin(\theta),X0)$$
$$\widetilde{m_1 \cdot m_2} \rightarrow \widetilde{m_1} \cdot \widetilde{m_2} + -(\widetilde{m_2} \cdot \widetilde{m_1})$$
$$rot(\widetilde{ax,\theta}) \cdot m \rightarrow rot(ax,\theta) \cdot m \cdot t(\widetilde{rot(ax,\theta)})$$

First we introduce the following notations :

$$R_1 = rot(Z0,theta1) \quad \& \quad R_2 = rot(Z0,theta2)$$
$$c_1 = cos(theta1) \quad \& \quad c_2 = cos(theta2)$$
$$s_1 = sin(theta1) \quad \& \quad s_2 = sin(theta2).$$

Now the term :

$$-tilde(R1 \cdot mul(r_{12},X0)) \cdot Z0 + tilde(R1 \cdot R2 \cdot mul(r_{23},X0)) \cdot Z0$$

is simplified (given properties) to

$$-(mul(r_{12},tilde(R1.X0).Z0)) + mul(r_{23},tilde(R1.R2.X0).Z0)$$

and is simplified (given properties and rules) to

$$mul(r_{23} * (c_1 * s_2 + c_2 * s_1) + -(r_{12} * s_1),X0)$$
$$+mul(-(r_{23} * (c_1 * c_2 + -(s_1 * s_2))) + r_{12} * c_1,Y0)$$

In this example one can notice that most of the rules compute formal evaluation, for example the rule $t(Id) \rightarrow Id$ "evaluates" the transposition function in the point Id. The declared properties for $+, -, *, inv, \cdot$ fit the standard behavior required for any simplifier.

2.4 Definition of complex properties

In our current implementation, the basic properties are associated to operator "by hand" using a function called declare. The operator "declare" is

associative and will be interpreted as "put all this information in the base". We want to introduce a sort of language by assertion in order to express **complex** properties. Then if the user asserts a complex property, the system computes all the basic declarations (in term of basic properties).

For example the complex property $group(f_1, f_2, f_3)$ can be expressed as :
$group(f_1, f_2, f_3) \rightarrow declare(associative(f_1), has\text{-}zero(f_1, f_3),$
$has\text{-}symmetric(f_1, f_2), involutive(f_2))$
From the previous definition of a group, the definition of commutative group is easy :
$commutativegroup(f_1, f_2, f_3) \rightarrow declare(group(f_1, f_2, f_3),$
$commutative(f_1))$
$commutative(f_1) \rightarrow declare(g - symmetric(f_1, Identity))$
Now if the user asserts $commutative\text{-}group(+, -, zero)$,
the system will rewrite it in $declare(group(+, -, zero), commutative(+))$
then in $declare(declare(associative(+), has\text{-}zero(+, zero),$
$has\text{-}symmetric(+, -), involutive(-)),$
$declare(g\text{-}symmetric(+, Id)))$
then because "declare" is associative
in $declare(associative(+),$
$has\text{-}zero(+, zero),$
$has\text{-}symmetric(+, -),$
$involutive(-),$
$g\text{-}symmetric(+, Identity)).$

We think that this is an easy and concise way of giving properties to operators.

3 Defining an operator by evaluation functions

In existing systems (except Scratchpad), evaluating a term consists in calling the evaluation function attached to the root operator of the term on its arguments. Each evaluation function is associated to an operator name and tests dynamically the types of its arguments. The code of the function that adds two arguments is essentially the lisp function :

```
(de add (e1 e2)
    (selectq (type-of e1)
             (integer
              (selectq (type-of e2)
                       (integer (add-integer e1 e2))
                       (polynomial (add-int-poly e1 e2))
                       (matrix (add-int-mat e1 e2))
                       (t (make-term + e1 e2))))
             (polynomial ... )
             (matrix ... )
             (t (make-term + e1 e2)))).
```

The user cannot modify the source of such functions, therefore he isn't able to define new semantics for those operators.

We point out another solution which consists in splitting those evaluation functions in the primitive ones (add-int-poly, add-integer, add-matrix) associated with their functional types ($integer \times integer \rightarrow integer$). A primitive evaluation function associated to its functional type is called an **interpretation**, and is denoted by ($integer \times integer \rightarrow integer$,*add-integer*). In this context, evaluating a term is consists in two separated phases : a type checking, and a call of the function if it matches.

The previous code of "add" is equivalent to a lot of interpretations :
($integer \times integer \rightarrow integer$, *add-integer*),
($integer \times polynomial \rightarrow polynomial$, *add-int-poly*),
($integer \times matrix \rightarrow matrix$, *add-int-mat*),
($polynomial \times polynomial \rightarrow polynomial$, *add-poly*),
($matrix \times matrix \rightarrow matrix$, *add-matrix*),
...

Generally, in computer algebra systems, conversions exist between types of objects, but they are implicitly encoded in evaluation functions. For example defining ($integer \times polynomial \rightarrow polynomial$,*add-int-poly*) is equivalent to defining ($polynomial \times polynomial \rightarrow polynomial$,*add-int-poly*) and the conversion from integer to polynomial. Defining conversion leads to decrease the number of interpretations to be defined. The user is given this facility but he can also give all the functions so that it doesn't increase time computation by computing conversions.

The previous code of "add" is now equivalent to only three interpretations :

$(integer \times integer \rightarrow integer,$	$add\text{-}integer),$
$(polynomial \times polynomial \rightarrow polynomial,$	$add\text{-}poly),$
$(matrix \times matrix \rightarrow matrix,$	$add\text{-}matrix),$
associated to two conversions :	$integer \overset{convert}{\rightarrow} polynomial,$
	$integer \overset{convert}{\rightarrow} matrix.$

3.1 Evaluation

In order to evaluate a term, the system considers all the interpretations. For each one, it applies the function every time a generalized sub-term (definition in the section2.1) matches the functional type.

Example : The following interpretation is attached to the operator + : $(integer \times integer \rightarrow integer, add\text{-}int)$.
To evaluate $2 + n1 + 3 + n2 + 4$, the system considers the interpretation, search a generalized sub-term that matches the functional type. It finds the generalized sub-term $2 + 3$, computes $apply(add\text{-}int, 2, 3) = 5$ then tries to evaluate $n1 + n2 + 5 + 4$. This last term is evaluated in the same way to $n1 + n2 + 9$ which cannot be evaluated again.

3.2 Evaluation modulo conversions

We can see the relation "can be converted to", denoted by $\overset{convert}{\rightarrow}$ as an ordering between types, $T_1 \leq T_2$ means $T_1 \overset{convert}{\rightarrow} T_2$. Then we can define equivalence classes between types by $class(T_1) = class(T_2) \Leftrightarrow T_1 \leq T_2$ and $T_2 \leq T_1$. We call **numerical types**[10], the types used to define functional type, and **formal types**[11] the equivalence classes associated to $\overset{convert}{\rightarrow}$. For example if two kinds of integer called I_1 and I_2 are implemented, with $I_1 \overset{convert}{\rightarrow} I_2$ and $I_2 \overset{convert}{\rightarrow} I_1$, they belong to the same class, which can be called $Finteger$.

Declaring a conversion between two numerical types means to declare a conversion between each couple of types of their two classes. For example given the class $(Integer, I_1, I_2)$ and the class $(Polynomial, P_1, P_2)$, the declaration $I_1 \overset{convert}{\rightarrow} P_1$ is equivalent to $I_1 \overset{convert}{\rightarrow} P_2$, or $I_2 \overset{convert}{\rightarrow} P_1$ ... So when

[10] A numerical type is a name associated to a representation.

[11] A formal type is a name associated to several numerical types, then several representations.

the formal types are defined, it's enough to declare the one-sided conversions between them.

To evaluate a term modulo the interpretations of its root operator, the same algorithm is used. The only difference comes from the matching function between a sequence of terms and a functional type.
In this context, a sequence of terms $t_1, ..., t_n$ matches a numerical functional type $T_1 \times ...T_n \rightarrow T_{n+1}$ if the numerical type of each term in the sequence $ntype(t_i)$ can be converted in the corresponding type T_i in the functional type.

In the presence of conversions, the system applies the evaluation function in a different way than before : it first computes the conversions of the arguments, then applies the right function. In order to convert a numerical object the system has only to convert it to a term (it writes it : i.e. this is equivalent to the pre-processing applied before the display of an expression) and then has to convert this term to a new numerical object (it reads it). We notice that if it is not the most efficient way, it's the most easy and general one.

4 Interactions with an host system

Basically, beside the library, a computer algebra system consists of three components a parser, a simplifier and a display process. If the simplifier is completely customizable, the two other ones must be also customizable. For example, if "+" is declared associative, one may want it to be infix and n-ary ...
Then the base contains algebra knowledge (properties, interpretations) as well as syntactical knowledge (infix, prefix, postfix, matchfix, right and left priorities ...) associated to operator names.

In existing systems, numerical types are predefined. Even if one was able to define new types, they couldn't be taken into account. In our system, numerical types are free, but they must be declared. According to an object oriented approach, to define a numerical type, one must give a name (T), and some fields ($a_1, ..., a_n$). But this is not sufficient because the system needs a function to read an object of type T, and another one to display it. Those functions are used in order to compute numerical conversions (see 3.2).

5 Final remarks

At this time, such a simplifier is implemented in Le-Lisp with a parser and a display function similar to those of the computer algebra system Sisyphe ([GGP90]). A base has been built which uses Sisyphe as a library of lisp functions. The only difference between this system and Sisyphe itself comes from the lack of error detection. For example one can write *trace*(3), the system will probably not find any interpretation to *trace* over integers, and the output term will be the same as the input term. We think that some users may need to raise type errors, so a formal type-checker is in the course of implementation. The user will have to associate all the authorized formal (with formal types) functional types to each operator. Type checking will be computed only if the user requires it.

The simplifier we have described can be seen as an advanced interface between the user and a library of algorithms. In order to have the term $op(x, y, z)$ simplified by the system, the user must link *op* with a function f in the library, and its numerical functional type (we did so to connect our simplifier and the system Sisyphe). If f has got implicit properties he must associate them to *op*, furthermore he must declare the numerical types (names and structure), and the conversions. It's an advanced interface because, first it doesn't compute only over numerical objects but also over formal ones (when the formal type check will be implemented), secondly the action of this interface is completely defined by the user. Finally, given an engine, many simplifiers can be created modifying the knowledge base and linking it to an existing computer algebra system.

References

[CG89] Y. Papegay C. Garnier, P. Rideau. Modelisation dymamique litterale. In *Computer methods in applied mechanics and engenierring*, pages 215–225, Elsevier Science Publishers B.V., Versailles, 1989.

[GGP90] André Galligo, José Grimm, and Loïc Pottier. The design of SISYPHE : a system for doing symbolic and algebraic computa-

tions. In A. Miola, editor, *LNCS 429 DISCO'90*, pages 30–39, Springer-Verlag, Capri, Italy, April 1990.

[Hul80] JM. Hullot. *Compilation des formes canoniques dans des théories équationnelles.* PhD thesis, Université de Paris-Sud (centre d'Orsay), 1980.

[JSW86] Richard D. Jenks, Robert S. Sutor, and Stephen M. Watt. *Scratchpad II: An Abstract Datatype System for Mathematical Computation.* Research Report RC 12327, IBM, November 1986.

[The83] The Mathlab Group. *Macsyma Reference Manual, version 10.* MIT edition, 1983.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399